Hans Jürgen Holstein

HOMO CYBERNETICUS

3

Heuristic Simulation Programming

UPPSALA: SOCIOGRAFICA. 1974

Acta Sociocybernetica Upsaliensis

Digitized by the Internet Archive
in 2026

https://archive.org/details/IA41548924_0009

Hans Jürgen Holstein

HOMO CYBERNETICUS

VOLUME THREE

Heuristic Simulation Programming

UPPSALA: Sociographica, 1974

Copyright (C) by Hans Jürgen Holstein

Preface

Homo Cybernéticus comprises five volumes. All are relatively self-contained, but complement each other into one whole. The *first* volume is a synopsis of the simulation intentions and approach adopted by us; its tacit main question is: "How can we operationalize these intangibilities?". The *second* volume tackles this problem on the methodological level by trying to elaborate a systematic epistemology for dynamic causal modelling. Its conclusions support our initial 'hunch' that simulation is the optimal tool for dynamic causal modelling. Volume *Three* is therefore an investigation into our possibilities of constructing a simulation tool (a computer program) which will allow us to concentrate on our problem instead of on the computer. We do not wish to adapt our problem to the computer, but expect the computer to adapt to our problem instead, so to speak. Volume *Four* leaves all computer problems behind and asks again how we can simulate human behavior; the investigation is this time less superficial than in the first volume. The *fifth* volume, finally, delineates a framework for rendering the causal simulation of societal macro-processes possible. It is simultaneously a first example of how the principles discussed in the earlier volumes can constrain, generate, and guide large-model concretizations.

The tackled task is vast, and although everyone agrees that it needs to be tackled, there seems to be a general suspicion that those who do it are charlatans and/or self-appointed prophets. It has hitherto been difficult to argue for *Homo Cybernéticus* properly because its 'point' lies in the linking together of a plurality of diverse fields of research,

and also because it contains relatively much 'New Think'. Especially the advocation of common-sense thinking at the expense of statistics has evoked skepsis and territoriality defenses. Sociologists have thus downgraded it all to 'mere' computer science, while computer scientists have considered the simulation project oddly humanistic and too problem-oriented.

I was therefore very grateful when the Swedish Riksbank Tercentenary Fund dared to give financial support to my research in 1968. It has done so until this day.

The other important event was the foundation of the Computer Science Laboratory in Uppsala by Dr. Erik Sandewall and myself, as a 'home' for our up to th then 'homeless' research projects, his being in the field of artificial intelligence, mine in socio-cybernetics. This symbiosis has, at least for my part, been immensely stimulating.

I thank also my collegues *Lennart Stålberg*, *Torgny Tholerus*, *Tore Risch*, and *René Reboh*, most sincerely for their participation and cooperation in this sociocybernetic venture during the past years.

Uppsala, in April 1974 H.J.H.

Table of Contents

PART I: *The Atman-Design Principles*, 1

1. Introduction: Atman Monitors, 3

2. Monitor and Model, 6

3. The Database, 8

3.1. Contexts, 9
3.1.1. Quasi-Physical Contexts, 10

3.2. Entities, 11

3.3. Properties, 12

3.4. Database Generation, 13

4. The Operation Rules, 14

4.1. The Default Model, 15

4.2. Jigs, 16
4.2.1. Jig Properties, 17

4.3. Jig Sequences, 21

5. The Task Stack, 22

6. The Top Loop, 23

7. Jig Execution

7.1. Execution Conditions, 22
7.1.1. Tacit Execution Conditions, 26
7.1.2. Explicit Execution Conditions, 27

7.2. Execution Exit Values and Execution Results, 28

7.3. Execution Statistics, 29

8. Time, 30

9. Conclusion, 31

PART II: *Atman in LISP ('Atman2')*, 33
(Written in collaboration with Tore Risch.)

10. Atman2 Components, 35

11. The Top Loop, 35

11.1. The Jig Processor, 38
11.1.1. Jig Execution, 39

12. Jig Definition, 41

13. Pushjig, 42

13.1. Repushing, 44

13.2. Autopushing, 45

14. Jig Sequences, 46

15. Interaction, 48

15.1. Feedback, 50

16. Execution Control, 51

16.1. Nogo, 51

16.2. Continue, 51

16.3. Xqcontrol, 52

17. Time Points and Time Steps, 53

PART III: *Atman in REC ('Atman3')*, 55

18. Introduction, 57

19. Basic Rec, 58
19.1. REC and the Variable-Free Calculus, 59
19.2. The REC Semantix, 61

20. Atman-REC, 70
20.1. Jig Definitions, 71
20.2. Jig Sequences, 72
20.3. Executions Suspension, 74

21. Syntax Extensions, 75
21.1. Argument Indocators, 75

22. The Atman3 Database, 77

23. Hypothetical Reasoning, 81

PART IV: *Database Generation*, 83

24. Database and Morphologies, 85

25. The Kinship Morphology, 86

26. Attributes, 91
26.1. Ages, 91
26.2. Social Status, 93
26.3. Attribution in Atman2, 94

27. Physical and Social Morphology, 97

28. References, 113

PART I: THE ATMAN-DESIGN PRINCIPLES

1. Introduction: Atman Monitors

We have developed a design for program-control flows which will enable us to write programs that can generate the complex processes we intend to model, namely everyday life. The exact implementation of this design in program form can be more or less elaborate, elegant or efficient, and we shall here not be concerned with that. (But this booklet includes two such implementations in different, although similar, languages.)

We call this program-control-flow design *atman monitor* and regard it as a very abstract model of how the world is structured and how it behaves. It is a way of translating parallelity of processes into sequential form, and also a manner of dealing with the multiplicity of description levels we are used to in ordinary reasoning and communication.

We have attributed utmost importance to the intiutiveness of modelling. That is: we have been determined to keep on rejecting modelling designs that one not immediately responds to as obvious. (In consequence, we have learned that 'the obvious' sometimes is hard to spot and trace.) If most people regard a comprehensive model as obvious in its structure and as trivial to use, it must be a very good model, we reasoned.

Of course, we have not yet gotten quite that far, but perhaps finally far enough to be able to evoke some interest in the cooperative search for the very good model. (There cannot reasonably be very many significantly different models of the kind we look for, that is why we do not speak of the search for *models* in the plural.)

We envisage an ultimate model where we can converse with the simulated individuals in natural language, and where they understand the semantic as well as the pragmatic meaning of what they 'say' and of what we tell them. Such a model will then only exceptionally require special programming abilities from us. We estimate that such a model can be realized within the next decade.

If hundreds and thousands of program pieces are to be put together into a single coherent and consistent whole, it is of prime necessity to facilitate this composition. The main obstacle to the realization of an atman simulation system during the sixties has surely been our inability to organize the total-model composition in a humanly feasible manner. It seems that all the bits and pieces which are going to constitute the first model version have been known for a long time, and that only the very vision of how to cope with the heap of details was lacking. *Homo Cybernetius* is such a vision although the perfection of its implementation, in the shape of an atman simulation system, may take another decade.

Volume 1 gives ample background information on the considerations we had decided to take while we searched for our design for a simulation monitor. Volume 4 illustrates how the monitor can be used for non-trivial model synthesis. This present volume focuses on the technical aspects of that part of the simulation system which, by the average user, eventually can be regarded as a black box which he needs to worry as little about as he worries about the intestines of the computer at his local computer center.

In order to be able to make the monitor a black box, we need to minimize its specificity. Whatever we believe to be easily variable, is to be specified outside the monitor. The monitor is merely to organize the bits and pieces of our model specifications into a coherent whole and to relieve us of the drudgery of spelling out all the mere technicalities that have nothing to do with the modelled problem, but are associated with the modelling method.

2.Monitor and Model

The essential characteristics of simulation is the recursive application of a set of rules ('the operation rules') to a database. The database that we start with is usually called *initial state*; the sequence of results of operation-rule applications is called *the process generated by the model*. The simulation model consists of the database and the operation rules; but the rules that govern the application of operation rules to a database in general are not part of the simulation model (at least not on the same logical level); these latter rules are called *meta-rules* or *monitoring rules*, or simply *monitor*. (The simulation model is thus the very database to which the monitor is applied. This suggests the possibility of constructing a meta-monitor which manipulates the monitor.)

The monitor can be equipped with one or several subsets of rules that govern the structuring and/or the default valuation of the initial database according to given superficial indications by the user. ("Default valuation" means the assignment of (reasonable) values to variables that have not been explicitly been given values by the user.) We envisage two types of such 'structure monitors': those that are used when no empirical data are given and those that govern the entering of specific empirical data into the database. The two types are not mutually excluding. The first type must 'invent' reasonable data, while the second type usually only 'writes over' *parts* of the total 'invented' database.

It is furthermore possible to let the monitor include sets of 'interpretation rules' that work on the database without changing it (except possibly

by explicitly adding derived database implications in order to facilitate their use). Such 'interpretation rules' could, for instance, be used for the synthesis of consise reports on states and processes in the model, to be given to the user or to simulated individuals, upon request or systematically.

3. The Database

The micro-structure of the database is irrelevant for the (general) monitor design. Whether all information is stored in tables, in property lists, in trees, or in networks, is merely of methodological importance since the choice of representation structure can have repercussions upon the *ease* and *effectiveness* of implementations of various design features. The essential for us is the *meaning* that we ascribe to specific component structures within the database (to a specific network part, for instance). When we speak of a simulated individual's properties, we do not care how they are stored, because the manner of storage is (ideally) not to have influence upon the simulation outcome.

3.1. Contexts

It is typical of human reasoning that it uses categories, categories of categories, et cetera, and that such categories sometimes are treated as if they were individual objects themselves. This particularity must be reflected in the monitor design: it must allow the simultaneous use of several levels of aggregation, and it must permit the creation and deletion of aggregations during the simulation run; each aggregate must, if and as long as we wish, be treatable as an individual that behaves according to the same or to different principles in comparison to members of any category of individuals in the model.

We shall call these aggregates or categories *context* because we wish to accentuate the fact that category membership usually is conditional and temporary. A house, a bus, a story, a family, a bum begging for a dime, a project, all of these can be contexts. Some of them will be *physical* (houses and busses, for instance), others *social* (a family or a beggar and his victim, for example), others *imaginary* (a story or a project).

An individual is to be the member of exactly one physical context at any given point of time, but he can belong to arbitrarily many social or imaginary contexts simultaneously. Social contexts can *overlap* with physical ones; that means that the entering or leaving of one implies (or tends to imply) the entering or leaving of the other. Social contexts can, of course, also overlap with other social contexts, and they can also overlap with imaginary contexts.

Physical contexts can never *overlap* with other physical contexts, but they can *contain*, or *be contained in*, other physical contexts. Containment differs from overlapping in its exclusiveness: if an individual enters a contained one, he automatically leaves the containing context. It is thus guaranteed that he will never be in more than one place at a time.

This reflects ordinary reasoning: Assume that twenty persons get on a bus, sit down, and do not move around while the but takes them from one place to another. The bus is contained in a certain physical context (a town, for instance), and it contains the individuals. As the bus moves, we need not move the individuals, because they automatically follow along in the context in which they are contained (the bus, namely) and they do not (but they could) change their relative positions within the context that contains them(the bus). We avoid thus the ominous 'frame problem' at least for spatial translations (cf. Hayes 1973) and obtain furthermore a very efficient simulation of collective transportations.

3.1.1. Quasi-Physical Contexts

It is not unusual that we employ spatial conceptions when we speak of non-physical things. A party fraction that moves to the left, or a class which income-wise catches up with another, are examples of such 'figures of speech'. It is desirable that we can maintain these conceptions also in the simulation model, and we must therefore be permitted to specify arbitrarily many quasi-physical 'property spaces' in which we can locate individuals and for which the containment principles (with all their implicit practical advantages) are valid.

3.2. Entities

We use the word "entity" to denote the class of individuals and contexts in the database. Entities are the 'agents', the dynamic decision units in our simulations. Entities are categorized, and each entity category is associated with a set of rules in accordance to which all members of the category behave. For example, persons behave in one way, busses in another, task groups in a third.

At any moment in time, we focus upon a single decision unit (person 17, or buss 14, or task-group 5) and refer to it as 'Self'.

3.3. Properties

All entities (that is: individuals and contexts) in the database can be given characteristics ('property values'). A property is simply a variable, a property calue is a value given to the variable. It is common that properties have restricted value ranges; we can, for instance, stipulate that the values of the property *age* are positive integers, or that the values of the property *memory-content* must be of the forms "(name number list)" or "(number name name)".

The *property catalogue* of an entity is the totality of all properties of that entity that are stored in the database. The property catalogue must minimally comprise all properties that are used as parameters of any operation rule that is to be applicable to the entity. The concrete property catalogue implies namely the entity's individuality if the property values are allowed to influence operation-rule applicability and/or the results of operation-rule applications.

3.4. Database Generation

A database can either be specified 'extensionally' by listing all entities in it along with the complete property catalogue of all entities, or 'intensionally' by giving rules which generate entities and other rules which define the property-value ranges for the various sorts of entities. A special generator is necessary in the second case; the generator (which is a set of rules) picks (usually at random) a value from the value range of each property and stores it as the property's value. The generator translates thus intentional definitions into extensional ones (cf. Volume 2, sec. 2.2.3 and 4.5).

For small databases will the intensional definition be more difficult to achieve than an extensional one, but intensional definitions do more than merely enable us to generate a database; they have also a heuristic value *per se* since we have a chance to detect regularities in the data, and to accumulate knowledge about such regularities. It is therefore worthwhile to struggle with intensional definitions. In fact, a slightly inadequate intensional definition is, in the long run, and for the reason of accumulation, better for science than a quite correct extensional one. Particularities can rarely be regarded as scientific processes.

Raw empirical data are, of course, always extensional definition components; data analysis is an attempt to derive (usually in a systematic manner) intensional-definition components from them. (Data analysis is thus the inverse of the above-mentioned generator.) Analyzed as well as unanalyzed ('raw') empirical data ought to be acceptable to the database-generating monitor part.

4. The Operation Rules

In order to be of any practical use, the monitor must be able to assist us in composing a model from many components. Experience shows that we rarely have a complete and well-organized model to start with; and it is self-evident that we have only vague notions when we begin to study a new problem. A simulation monitor must therefore take incompleteness, vagueness, and a desire to supply further specifications later on, into account. Also the order in which information is supplied ought to be largely irrelevant.

Atman monitors will therefore presuppose that the operation rules are supplied to them in the form of modules (called *jigs*); these modules will comprise indications regarding how they are to be linked, and the monitors will link them in the indicated manners. Jig specifications can be made *incrementally*, that is: even in the course of the simulation.

4.1. The Default Model

We assume that it must be most attractive for a new user to have access to a *default model*, that is: a model which is comprehensive but not very detailed (with some illustrative exceptions), and which he can use for orientation and to start with. It will presumably be significantly easier for a user to modify a given model than to specify a similarly comprehensive model from scratch.

If one wants to be able to associate default models with atman monitors, one must again see to it that incremental model modifications and/or elaborations are easy to achieve. The default model can then be used as a *map* or a *frame* for orientation; it will then not be difficult to find out how to link one's own model into the default model.

4.2. Jigs

Our concept 'jig' is meant to allude both to "template" or "stereotype" and to "jig-saw puzzle". Our jigs describe behavior stereotypes, and they are also small components in a large 'picture' of social reality.

Jigs are essentially subroutines or functions. Their only peculiarity is that they can be ascribed properties and that these properties govern their evaluation. In fact, during a simulation run there will be many copies made of these jigs; these copies are called *instances* and each will have individual properties.

Jigs (and thus also their instances) describe activities. Jig instances are always associated with certain database entities (individuals or contexts); when an instance is evaluated, it thus describes a current activity of the entity.

For example, let JNR be an instance that describes the activity 'sleep', and let JNR be associated with an entity of type 'person', called *Self*. When the monitor, following its queuing order among the instances that are to be evaluated ('executed'), happens to execute JNR, we can say that Self sleeps.

The *body* of a jig is the sequence of functions or rules that define a possible activity of an entity. Such a sequence describes in utmost detail what is to happen to the database if a person is to go shopping, or to enter a bus, for example.

4.2.1. Jig Properties

We distinguish five types of jig properties. Let "JNR" denote the currently focused instance.

(a) The first type comprises properties which the user can employ for governing the evaluation ('execution') of an instance. These properties are (minimally):
(a1) *Duration*; this property specifies the 'life time' of an instance ('JNR') once its execution has started. For instance, if JNR describes the activity *sleep*, we can let the duration of JNR (that is: of this activity) be 8 hours. (The default value is 1 time unit.)
(a2) *Attention*; this property specifies the amount of attention it requires from the simulated individual who performs the activity described by the instance. We assume that each individual has only a limited resource of attention to spend.
(a3) *Priority*; this property governs the order in which the instances are to be executed.
(a4) *Zeigarnik*; this property modifies the priority of an instance that has a duration which is greater than 1 time unit. We can thus specify that an activity is to have higher, lower, or unmodified priority once it has been started. (The default value is 0.)
(a5)*Xqevery*; this property is only specified if one wishes to avoid unnecessary executions of jig instances with durations that exceed 1 time unit. For instance, while Self sleeps we can choose to execute the instance of *sleep* only once every half hour (merely to give Self a chance to awaken), and we can perhaps save some computation time. Default value is 1, which means that JNR is executed each time it is tested.

(a6) *Interruption*; it can happen that a jig instance's execution must be interrupted. The property 'interruption' specifies rules to be followed in case an interruption occurs. Default value is "abandon". (Cf. section 7.1.1.)

(b) The second type of jig properties comprises properties that are automatially valuated (that is: given a concrete value) by the monitor on the basis of the situation in which the instance JNR was created:
(b1) *Created*; this property specifies the exact time at which the instance JNR was created.
(b2) *Master*; this is the name of the 'master jig' of which the instance JNR is to be a copy.
(b3) *Executor*; this is the name of the entity which is to behave according to JNR's master's activity description.
(b4) *Arguments*; this is a list of the arguments (if any) given to this instance JNR when it was created.
(b5) *Ancestor*; the creation of new instances happens always during the execution of another instance (except initially in the run), and JNR's property "ancestor" has as its value the name of the instance that was under execution when JNR was created.
(b6) *Initiator*; this is the name of the entity which was the 'executor' of the 'ancestor' of JNR.

(c) The third type comprises properties that are automatially valuated during execution of the jig instance:
(c1) *Xqstarted*; this is the point in time at which JNR's execution happened to start.
(c2) *Xqlatest*; this is the point in time at which JNR last was executed.
(c3) *Descendants*; this is a list of all the instances that have JNR as their 'ancestor'.

(d) The fourth type comprises properties that can be used for transferring jigspecific information from one instance to another, or from one point in time to another. We have as yet found the following jig properties to be useful:

(d1) *Xqcontrol*; if another instance has given JNR a value for JNR's property 'xqcontrol', JNR is not to be executed but to return the value of JNR's property 'xqcontrol'. We can thus stop, trigger, or suspend other instances.

(d2) *Succedes*; this is a list of all the instances that must be completely executed before the execution of JNR can be started.

(d3) *Precedes*; this is a list of all the instances which cannot start before JNR has been executed completely. When this is the case, the name of the now executed instance JNR is removed from the 'succedes'-lists of all the instances in the 'precedes'-list of JNR.

Another property can be used for pushing recurring tasks:

(d4) *Autopush*; this property has a value of the form "$(t_o\ s_1\ s_2\ s_3\ \ldots\ s_m)$". "$t_o$" refers to the time point when the jig instance is to be executed for the first time; "s_i" refers to a time interval (= a non-zero number of time units) after which the jig instance is to be repushed into jig queue for its i+1st execution. s_{m+1} is equal to s_1.

(d5) *Stopautopush*; this property's value is a number which indicates after which time point JNR no longer is to be autopushed.

When an autopushing jig instance JNR is to be abandoned forever, all its decendants are also to be abandoned forever. (This rule can easily be overriden by changing the value of 'Descendants' during execution.)

(e) The fifth type of jig properties comprises all other properties one wishes to give a jig or jig instance, but which do not govern the monitor's behavior.

4.2.1a. Default Values of Jig Properties

In order to minimize redundancies while programming jigs, we let all jig properties of type *a* assume default values in case no explicit values have been specified by the user.

In fact, if the user did not explicitly associate a type-a value with a jig instance JNR when it was created, there are *three* levels of default values. First, the particular 'executor' of JNR may store a particular value for instances of a particular jig. Second, the master jig may have a value for this property. Third, let there be a non-specific default value for all jigs without specified values for this property.

4.3. Jig Sequences

A *jig sequence* is an ordered set of jig instances. The ordering is achieved by ascriptions of the jig properties 'precedes' and 'succedes'. The elements of a jig sequence need not belong to the same task stack.

The priorities of its predecessors must be at least equal to that of any sequence element we regard. Adjustments must be made automatically by the monitor, to ensure that a high-priority task is not blocked by what usually is a triviality of very low priority.

JNR is able to cancel its own execution if not all prerequisite conditions are fulfilled and to add subtasks (which are to attempt to achieve the required state of affairs) 'in front' of itself in the jig sequence.

JNR can also create parallel jig sequences or to insert instances somewhere among its successors in the sequence.

5. The Task Stack

Each entity (decision unit) has a 'task stack' (we call it also 'jig queue'), that is a set of FIFO-ordered lists of jig-instances which are to be 'executed' by this entity. The total jig queue is partitioned into m (for example 10) subqueues, each associated with a certain 'priority' value. Thus, an instance with priority value m will be put into the first of the FIFO-ordered subqueues, an instance with the priority value 1 will be put into the last one.

Neither the moment of pushing nor priority are thus sole determinants of the order of 'execution' of the pushed jigs (that is, of the jig instances) of an entity. ("To push" jig J means "to put an instance of the master jig J into one of the subqueues of the jig queue of an entity".)

(Remark: "FIFO" is short for "first in, first out". Its contrast is denoted by "LIFO", which means "last in, first out".)

6. The Top Loop

The Top Loop is a circular sequence of monitoring rules which focus on all 'Selfs' (that is: on all entities in the database that belong to categories for which we have defined jigs) in a user-determinable order. The variable called *focusorder* contains an ordered list of all entity categories we wish to focus upon; such a list can, for example, be "(person family bus village region)". Unmentioned entity categories will contain static entities because no behavior is triggered. But *focusorder* can be made to change value and we can thus include certain relatively immobile entity categories now and then in the list. The list order determines the focusing order.

All members of a focused category are focused in random order (to avoid systematic errors stemming from a particular focusing order). Randomized focusing is meant to simulate parallel focusing; all entities belonging to the categories mentioned in the focusorder-list are to be regarded as behaving in parallel. One scan through the focusorder-list is assumed to take exactly one time unit. That means that each entity is focused *exactly* once during each time unit. (We can, in practice, make a random mix also of the jig instances (with the same priorities) from the jig queues of all entities in a category, in order to simulate parallelity still better, but we shall ignore this slight refinement here for ease of description.)

Each entity 'executes' a number of jig instances while it is focused. (While it is focused, it is referred to as 'Self'.) How jig instances are selected for 'execution' and how we determine when to

shift focus to another entity, is described in the next section.

7. Jig Execution

"Jig execution" (and more precisely:"jig-instance execution") refers to the evaluation of a jig body, that is: of the rules associated with a jig. This evaluation is parametrized and often conditional; that is, before a jig instance can be executed, a number of conditions must often be fulfilled, and when it is executed, its behavior is governed by jig properties (that is, more precisely: jig - instance properties), Self's properties, and environment properties.

7.1. Execution Conditions

When a jig instance is handled by the monitor, it is not certain that it is executed. It is *tested* first whether the execution conditions are fulfilled. Some of them are tacit, some of them explicit.

7.1.1. Tacit Execution Conditions

The first execution condition to be tested is whether JNR's 'Xqcontrol' property is bound to a value. If yes, this value is returned. (Cf. sect. 4.2.1, d1.)

The second execution condition to be tested is whether JNR's 'Xqevery' property suggests the execution. If not, JNR's execution is postponed.

The third execution test is for whether JNR's 'duration' has expired. If this is the case, JNR is abandoned (that is, deleted unless it is to be autopushed later on).

The fourth execution test is for whether JNR is 'preceded' by any other jig instance. If this is the case, JNR is postponed.

The fifth execution condition to be tested is whether Self still has as much 'attention' left as JNR requires. If not, JNR's execution must be postponed. However, if JNR was under execution the moment before, an execution postponent amounts to an execution interruption, and if JNR is equipped with the name of an interruption rule (as the value of the jig property 'interruption'), that rule is applied. If JNR is part of a jig sequence

(that is, if it 'precedes' other jig instances), and JNR is to be abandoned if interrupted, a special indication is necessary which tells whether the whole jig sequence is to be abandoned. ('Xq-control' can be used for stopping the successors.)

7.1.2. Explicit Execution Conditions

After all tacit tests, if the instance JNR still is 'executable', the monitor will start to evaluate JNR's 'body'. Explicit execution conditions will be part of the jig body, and they will usually be the first rules in the sequence of rules that constitutes the jig body. If all conditions are fulfilled, the whole body will be executed and the result will be the value "Continue"; if some condition is not fulfilled, the monitor will stop JNR's execution and assume the execution-exit value "Postpone". The result "Postpone" implies that JNR still is regarded as never executed; the property *Duration* remains thus unmodified, *Zeigarnik* has no effect, and *Xpstarted* still has no value.

7.2. Execution-Exit Values and Execution Results

We regard as the 'proper' execution results all those effects of function evaluations which somehow have changed the database. Besides these 'proper' results we assume the jig bodies to 'point out' exactly one of the four socalled 'execution-exit values' "Postpone", "Continue", "Abandon", or "Occupied". Which value is pointed out by JNR depends upon which of the tests it has passed or failed.

If "Postpone" or "Continue" are pointed out, JNR is put back onto the task stack (but will not be tested once more at the present moment). If "Abandon" is pointed out, JNR will not be put back onto the task stack (which implies that it can be deleted).

"Occupied" is pointed out if Self is currently occupied with the execution of so many instances that there is no 'attention' left for any additional jig executions. JNR's execution is postponed and no more of Self's jig instances need to be tested for execution. (This requires merely that zero-attention ('auxiliary') jigs, usable for 'monitoring' purposes of any kind, have automatically been put into a reserved priority class which is acknowledged *before* the 'proper' priority classes are.)

7.3. Execution Statistics

The monitor ought to compute a few statistics which allow the user to control the pushing, the duration and the abandoning of jigs (instances) dynamically. For example, if the jig queue is very long, we might wish to suppress the pushing of low-priority jigs and choose shorter durations and abandonments more often. (This way we can simulate the entity's being stressed.)

8. Time

In atman simulations will new events be triggered by other events or by time. Time-step sizes are variable and time points are definable in the same fashion in which we use time-point references in real life.

A *time unit* is the interval of time which we choose to regard as the 'same' moment. A *basic time unit* is the smallest time unit we wish to use in a simulation, a minute, for example; other time units can be thought of as composed of basic time units. Time is first of all counted as the number of basic time units that have passed since the simulation started. Any number of aggregations (in hours, days, weeks, months, and years) can then be derived from the basic time count, and constants can be added to them (for instance, in order to write "1974" instead of "24" in a simulation of a social development since 1950).

A *time step* is an increase of the integer variable 'Now' by a positive number n. This arithmetical operation symbolizes the passing of n basic time units. The number n is called *current time-step length* (or 'fugit').

A *time point* is a value of the variable 'Now'. It can be referred to in relative terms (for example, by saying "5 hours from now") or in absolute terms (for example, by saying "on March 6, 1974, at 21:35").

9. Conclusion

The outlined design seems to us simple; we doubt that an essentially simpler monitor can be found. We have tried to find an optimal balance between a completely controllable monitor (that must be one without own characteristics; that is, it can be omitted) and a monitor that imposes a rigidly detailed logical structure upon all the models it organizes. We wished the degree of world-structure specificity to be controllable; the monitor embodies merely a minimal (abstract) world structure which we assumed to be sufficiently unquestionable to be put into the monitor's 'black box'. But we envisage a *system* of more specific monitors in the form of *default models*, which all use the general atman monitor, of course, but which themselves are specialized on monitoring specific categories of submodels. The ultimate goal is thus a hierarchy of monitors of various degrees of generality.

PART II: ATMAN IN LISP ('ATMAN2')

(Written in collaboration with Tore Risch.)

10. Atman2 Components

We can distinguish four relatively independent parts ('packages') within Atman2:

(a) GETT/PUTT, which is a 'package' of functions of use for the storing and fetching of information in property trees (cf. Risch 1972).

(b) RANDOM, which comprises functions for samplings of various kinds.

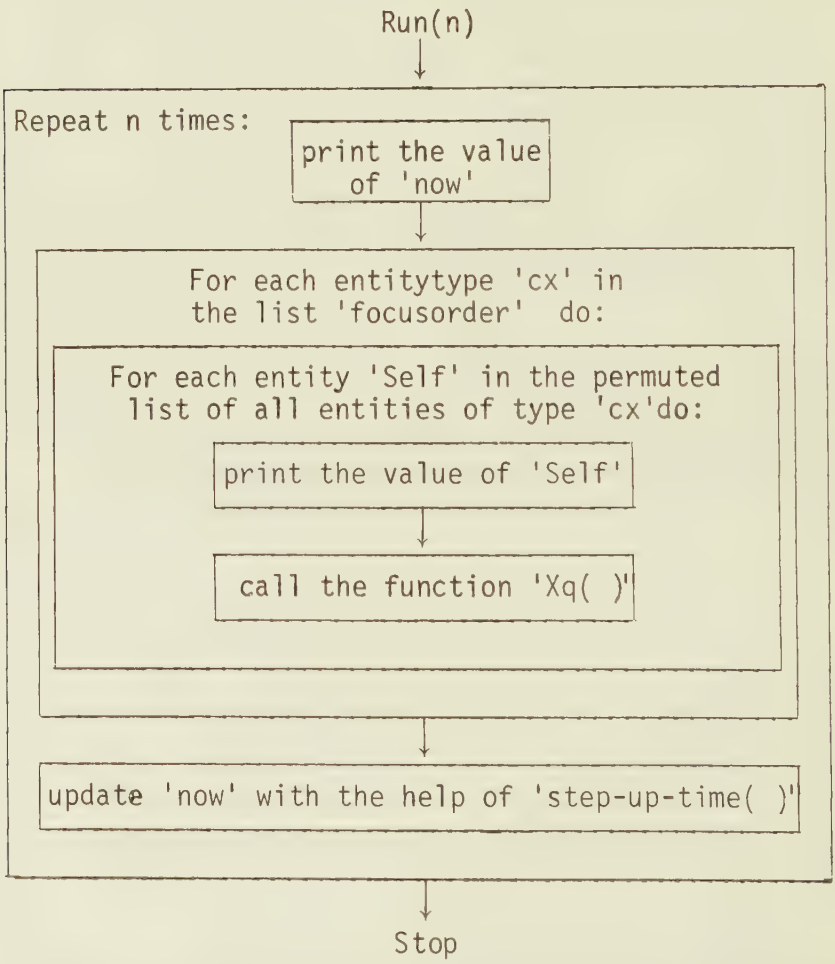
(c) CREATE, which comprises functions of use for generating the initial database (cf. Part 2).

(d) ATSIM, which is the set of central monitoring rules.

A package which furnishes macro-definition facilities (that is: enables one to define a private input language, tailormade for the problems one intends to formulate) has furthermore been employed (cf. Risch 1973), but is not a *necessary* program part.

11. The Top Loop

The flow chart for the top loop is as follows:


Run(n)
Repeat n times:
print the value of 'now'
For each entitytype 'cx' in the list 'focusorder' do:
For each entity 'Self' in the permuted list of all entities of type 'cx'do:
print the value of 'Self'
call the function 'Xq( )'
update 'now' with the help of 'step-up-time( )'
Stop

"Now" denotes the present time point. 'focusorder' is a list of entity types. 'Xq' is the jig process-

sor in ATSIM; it is iteratively applied to all instances JNR in the jig queue of Self.

11.1. The Jig Processor

When 'Xq( )' is called, 'Self' is already bound to the presently focused entity. At the database node 'Self::Taskstack' it finds the relevant jig queue which it will work on. But 'Xq' will first search through a list found at 'Self::Memory' in order to see whether any jigs are to be 'autopushed'(in the sense of Part I/4.2.1, d4) into the relevant jig queue.

This jig queue is thereafter transferred to the node 'Jigkew'. 'Self::Taskstack' obtains an empty list instead.'Self::Attention' is set to zero.

(a) The first element of 'jigkew' is bound to the variable (or node) JNR. 'Jigkew' is rebound to CDR of the list at 'Jigkew'. ("CDR of list L" means "the list L except its presently first element".)

(b) The function 'Xql' is applied to JNR. The execution-exit value returned by 'Xql'(cf. Part I/7.2) governs what is to happen with JNR after this application:
If "Postpone" is returned, JNR's execution is postponed.
If "Continue" is returned, JNR's execution is continued.
If "Occupied" is returned, Self is occupied with the execution of so many instances that there is no 'attention' left for any additional jig executions. JNR and the list at 'Jigkew' are put back onto Self's task stack (where also all postponed or continued instances have been put).
If 'Xql' returns any other value than these, JNR is abandoned.

(c) If "Occupies" has not occurred, and 'Jigkew' is not empty, go to (a), else return control to the top loop.

11.1.1. Jig Execution

The execution of JNR is performed by the function 'Xql' when it is applied to JNR. It 'behaves' as follows:

(a) If 'JNR::Xqcontrol' is not NIL, its value is returned.
(b) If 'JNR::Duration' is less than half of the value of 'Fugit', "Abandon" is returned.
(c) If 'JNR::Succedes' is not NIL,"Postpone" is returned.
(d) If 'JNR::Xqstarted' is NIL, it is given the current value of 'Now'.
(e) If the sum 'Self::Attention'+'JNR::Attention' exceeds 150, or if 'Self::Attention' exceeds 100 while 'JNR::Attention' exceeds 1,"Occupied" is returned.
(f) Otherwise, add the value of 'JNR::Attention' to 'Self::Attention'.
(g) If 'JNR::Duration' is NIL, or if ('JNR::Xqstarted' + 'JNR::Duration') exceeds 'Now', 'Xql' returns the result of the jig body's evaluation.
(h) If 'JNR::Xqlatest' is NIL, it is set to zero. If('JNR::Xqlatest'+'JNR::XqEvery') exceeds 'Now', "Continue" is returned without jig-body evaluation.
(i) Otherwise,'Xql' returns "Continue" after jig-body evaluation.

Explicit execution conditions can, during the jig-body evaluation lead to direct exits from 'Xql',

usually with "Postpone" as exit value. Such exists are thus not under the control of the jig processor but of the executed jig instance.

12. Jig Definition

A jig is defined by an expression of the form "(DEFJIG <name> <argument list> <list of jig properties>.<jig body>)"
where
<name> is the name of a LISP atom;
<argument list > specifies additions to the 'environment' in which the jig body is to be evaluated;
<list of jig properties> contains explicit value assignments to the jig properties; the form is "(P1 V1 P2 V2 ... Pn Vn)" where each "P" denotes a jig-property name, and each "V" an appropriate value; the "P" are not evaluated, but the "V" are;
<jig body> is a sequence of LISP functions; and the dot "." indicates that the extent of the jig body is not to be enclosed in separate parentheses; all following expressions are instead counted as part of the jig body. The dot is not written.

Example:

```
(DEFJIG GREET (X)  (ATTENTION (APPROX 20)
                          PRIORITY 8)
   (PRINT (LIST 'HELLO X))(PRINT
                          'HOW DO YOU DO)) )
```

The jig body is automatically 'macro-expanded', that is, the submitted code is modified such that defined abbreviations and other explicit syntactic formulation conventions ('macros') are replaced by their usually more complicated (but computionally more effective) correspondences. (Cf. Risch 1973.)

13. Pushing

(Jigs can in principle be used as if they were ordinary LISP functions; the atman monitor will then not govern them and no tacit execution conditions will therefore be tested. If there are any explicit execution conditions, however, they will be part of the jig body and therefore be evaluated with it. However, in the Atman2 implementation we have not made provision for such extra-monitor use.)

To submit a jig to the control of the atman monitor, we need to *push* it. Pushing a jig implies that the monitor makes a copy ('JNR') of it and attributes JNR with all explicitly given jig-property values, and with default values where necessary (cf. Part I/ 4.2.1a). JNR is then added to the jig queue of some entity, or, if it is to be 'autopushed' at a moment later than 'Now', to the 'memory' of that entity.

Pushing is commanded by expressions of the form "(PUSHJIGQ <J> <E>.<list of jig properties>)", where
<J> is itself an S-expression of the forms
"(<jig name> <arguments>)", where
<arguments> is zero, one, or several S-expressions, or
"(APPLY* <name identification> <arguments>), where <arguments> has the above meaning, and <name identification> is to evaluate to a jig name;

<E> is the name of a database entity; if <E> evaluates to NIL, its default value is the current value of 'Self'; and the dot "." and <list of jig properties> have the same meanings as in chapter 12.

The function PUSHJIGQ returns the name of the new jig instance as its value.

Examples:

```
(PUSHJIGQ (GREET 'Person 17) Self Manner 'Friendly
          Priority (Gdistance Self Person 17))
```

(If the returned value is the name "GREET 37", it indicates that the 37th copy of the jig GREET was created.)

```
(PUSHJIGQ (APPLY* (PICK1 '(GREET IGNORE))
                  'Person 17))
```

¶

If the created jig instance is found to have a non-empty list as its 'autopush' value, and the first element of that list evaluates to a number exceeding 'Now', the instance will be added to the 'Memory' of the entity denoted by <E>.

13.1. Repushing

An instance JNR can be repushed with priority P into the jig queue of the current Self, with the help of the expression "(REPUSHJIG JNR P)". If P is omitted, the value of 'JNR::Priority' is assumed instead.

13.2. Autopushing

No new jig instances are created when a jig is 'autopushed'; it is simply repushed. However, when the instance is to be abandoned (but to be repushed later automatically), all its jig properties of types 2 (cf. section 2.5.1) and 5, and the value of 'Xqcontrol', are removed.

14. Jig Sequences

Jig sequences are defined by expressions of the form
"(JIGSEQUENCE <type> <S>...<S>)"
where
<type> can take the values POST, PRE, or FREE, while
<S> is a LISP-expression which (under all conditions) ends up with a call of PUSHJIGQ.

Example:

```
(JIGSEQUENCE FREE
    (PUSHJIGQ (RISE))
    (COND [(IS-MAID)(PUSHJIGQ (MILKING))]
          [(IS-FEMALE)(PUSHJIGQ (PREPARE-
                                 BREAKFAST))]
          [T (PUSHJIGQ(INSPECT-STABLES))])
    (PUSHJIGQ (BREAKFAST)) )
```

JIGSEQUENCE returns the list of the created instances.

The three values of <type> have the following meanings:

FREE indicates that there are to be no succession or precedence links from JNR, the jig instance in which the jig sequence is defined, to any element of the sequence, or vice versa. (That is, 'JNR::Succedes' and 'JNR::Precedes' are not to be modified. That JNR is the 'ancestor' of the sequence elements, is recorded, however, as usual.)

PRE specifies that the defined jig sequence is (in its entirety) to precede JNR; that is: the last

sequence element's property 'Precedes' is a list in which JNR's name is an element, and 'JNR::Succedes' contains the name of the last element of the jig sequence.

POST specifies the opposite, namely that JNR is to precede the first element of the jig sequence.

No errors occur if we define sequences with only a single element. JIGSEQUENCE can thus be used to link single instances into a sequence in front or behind JNR. "(JIGSEQUENCE FREE (PUSHJIGQ (....)))" is, of course, synonymous with "(PUSHJIGQ (...))".

15. Interaction

INTERACTION is a 'macro' which, like a PROG in LISP (or like any ALGOL or FORTRAN program), is a sequence of functions, there can be 'labels' which one can 'go' to, and there can be several program 'exits'.

The syntactic form of INTERACTION is
"(INTERACTION <list of local variables> . <body>)"
where
<list of local variables> is a list of variable names which we intend to use within <body>; if the same name occurs elsewhere in the total program, that other occurrence has no effect upon the value of the local variable here, which merely happens to have the same name. That means that *if* we wish to refer to the value which a certain variable has outside of <body>, its name must *not* be found in the list of local variables;
<body> is a sequence of LISP expressions (to be specified below in greater detail); and
the dot "." has again the meaning it had in ch. 12.

Example:

```
(DEFJIG FOO (A B C) (PRIORITY 1 DURATION 1440
                                 XQEVERY 30)
      .
      .
      .
      (INTERACTION (X)
      L0 (SETQ X (F1 A B)) (COND [(NULL X)(GO L0)])
      L1 (F2 X) (IA-NEXT L3)
      L2 (F3 X) (IA-NEXT L3)
```

```
L3 (COND [(F4 C) (IA-NEXT L1)]
         [(F5 C) (IA-NEXT L2)]
         [(RANDOMLY) (RETURN)]
         [(RANDOMLY) (IA-NEXT L0)]
         [T (IA-NEXT L3)]))
   .
   .
   . )
```

L0, L1,L2, and L3 are 'labels' which either a 'GO-statement' or an 'IA-NEXT-statement' refer to. The important difference between "(GO L0)" and "(IA-NEXT L0)" is that the former means a program-execution continuation at L0 during *this* moment, while the latter also means a continuation of program execution at L0, but the *next* time FOO is executed by the atman monitor. "(RETURN)" indicates a definite exit from the interaction body. The values of the local variables are lost after a definite exit.

(IA-NEXT leads to a (non-definite) exit from 'Xql' with the value "Continue".)

We can write "(IA-NEXT Z)", where "Z" is no label but a variable which evaluates to a label.

15.1. Feedback

INTERACTION was designed to facilitate the description of interactions between several entities (but it can, of course, also be used for describing the behavior of a single entity over a period of time).

Interaction is often parametrized by *feedback* from others. "(REPLY X)", if written in an instance JNR, assigns the value of "X" to the jig property 'Answer' of the 'Ancestor' of JNR. The 'Ancestor' of JNR is in these cases often an instance containing an interaction expression, but feedback can also be of importance in jig sequences. "(ANSWER)", if written within the body of the 'Ancestor' of JNR, evaluates to the reply sent by JNR.

16. Execution Control

16.1. Nogo

The execution of a jig instance JNR must sometimes be inhibited because the test of explicit execution conditions in the jig body shows that execution is to be postponed (cf. Part I/7.1.2). The statement "(NOGO)" achieves an exit from the jig body with the value "Postpone"; 'Xqstarted' loses its value again and indicates thus that the execution of JNR has not yet started.

16.2. Continue

When the expression "(Continue)" is encountered in a jig body during execution, execution is suspended and "Continue" is returned as exit value from 'Xql'. This means that we can prevent an instance from being abandoned even if its 'Duration' has expired.

16.3. Xqcontrol

Assume that we wish to stop all other descendants of the "Ancestor' of JNR. The expression "(FORALL J IN (DESCENDANTS (ANCESTOR JNR)) DO

```
     (OR (EQ J JNR)
         (XQCONTROL J 'ABANDON) ))"
```

will achieve exactly this. (Cf. Part I/4.2.1,d1.)It will assign the value "Abandon" to the property 'Xqcontrol' of all J except JNR.

The expression "(STOPSEQUENCE J)" means that all jig instances which belong to the sequence to which also J belongs, are to be abandoned (with the help of XQCONTROL). If its successors are to be abandoned, one is to remove 'J::Succedes' first (and also the references to J from its precedents).

17. Time Points and Time Steps

Before one starts the very simulation process, one can define own time-unit labels; one needs, of course, indicate the number of *basic* time units comprised by each of the newly defined time units. This is done with the help of the functions

"(timesorts.<L>)" and "(timeprops.<L>)"

where <L> is a list of time-unit labels in the first case, a list of numbers in the second.

Example:

```
(timesorts minute hour day month year)
(timeprops    1    60   24   30    12 )
```

One sees from the example that we have imposed a restriction for ease of specification, namely that each (except the first) time unit's cardinality can be expressed as an integer multiple of the preceding unit's cardinality. (Remark: "Cardinality" means here "number of basic time units in a time unit".)

The example's specifications are tacitly assumed if one omits them in one's program. There are two global variables, 'timesorts' and 'timeprops' (that is, their names coincide with those of the two functions), which from the start have the values "(minute, hour, day, month, year)" and "(1, 60, 24, 30, 12)", respectively. It is these values we change by calling the above-mentioned functions. The current time-step length 'fugit' has the default value 1. The global variables 'minute', 'hour', 'day', 'month', and 'year' are set to zero from the start or when we call the above functions; observe that these variables are

not identical in meaning with the time-unit labels which look alike. "hour" as a label identifies a time unit, while "hour" as a variable (that is: when it is evaluated) denotes the current hour of the day. Also 'now' is set to zero from the start or upon a redefinition of the variables 'timesorts' and 'timeprops'.

The function call "(step-up-time)" updates 'now' by adding the value of 'fugit' to it. Simultaneously are (in the default case) 'minute','hour', 'day', 'month', and 'year' updated, if necessary. For instance, if the very first instructions in our program were "(fugit 63)" and "(step-up-time)", 'now' would be set to 63, 'minute' to 3, and 'hour' to 1.

In order to facilitate the specification of time intervals, such as "two years, 5 months, and ten days", we have defined a function called 'time' which is used as exemplified by:

"(time 2 'year 5 'month 10 'day)".

'time' expressions evaluate into the corresponding number of basic time units. "(time 1 'hour)" evaluates thus into the number 60; "(time 1'hour18'minute)" into 78; et cetera. However, by writing "(time 1 'minute)" instead of "1", and "(time 1 'day)" instead of "1440", the user becomes largely independent of the choice of basic time units.

PART III: ATMAN IN REC ('ATMAN3')

18. Introduction

The implementation of Atman2 has proven that the basic design principles are appropriate for facilitating flexible incremental simulation programming. The choice of implementation language is now no longer essential but a matter of convenience.

But we have also some new ambitions, and it is *not* inessential which programming language one uses during the design phase. We wish to obtain increased concision, transparency, and efficiency in our next implementation, and we wish also to prepare our database for automatic heuristic planning.

This demands a very modern programming language such as LISP 70 or QLISP (cf. IJCAI73). We have chosen REC, a successor language to LISP designed and implemented by Torgny Tholerus, which is unique in concision and consistency, and extremely easy to extend (and also to augment, it seems). The atman monitor has simply been embedded into REC as a part of REC itself. (We call the result therefore 'atman-REC' in our texts.)

Atman-REC will soon be extended further to fulfil wishes regarding transparency (that is, program readability). We feel that our programs ought to be almost completely self-documenting, which indicates that we envisage a REC-extension into the direction of (restricted) basic English.

We shall now describe basic REC. Atman-REC will not follow other rules that basic REC, of course, but already basic atman-REC functions will be complex compositions of basic REC functions and therefore more difficult to survey.

19. Basic REC

REC is based upon a variable-free theory of functions (cf. Tholerus 1972/10). The other specific characteristics (of the *basic* REC language) are (DLU74, p 42):

(a) There is a one-one correspondence between the REC program and the structure of its computer-internal representation. It is therefore easy to manipulate with programs, that is: to let programs modify or compose programs.

(b) There is a one-one correspondence between the REC program and a flow diagram that has been drawn according to very general notation rules to be specified below (19.2).

(c) REC has an operator syntax, which means that no separate input-language translation program is necessary. Basic-REC programs resemble Algol programs in form.

(d) REC is "go to" free. A go-to control jump is instead regarded as being a function call without parameters and without return. Each label in the program (resembling the go-to labels in Algol and Fortran) defines a new procedure.

(e) The left side of an assignment may be any REC expression.

(f) Functions can always also be regarded as automata (cf. Volume 2, chapter 3) that can be coupled in various ways. REC is uniquely well suited for depicting such couplings into the finest details. (We believe it to be superior to notations such as Iverson's (cf. Iverson 1962) with regard to readability.)

19.1. REC and The Variable-Free Calculus

REC is the only programming language based upon a variable-free notation; all other ones are based upon the lambda notation (DLU74, p 46). For example, assume that we wish to define a function h which adds the results of two other functions. In lambda notation:

$$\lambda(x,y)\ (f(x,y)+g(x,y)).$$

Variable-free notation allows you to write

f + g

instead. x, y, and z are *predefined* functions which pick the first, second, and third elements from their input lists and put them into their output lists (DLU74, p 46).

The input list of a notation is a list of 'values'. A 'value' is a piece of information, a number, for instance, or a name, or the definition of a function.

Also the output list of a function is a list of 'values'. A function is thus a 'device' which receives a list and which outputs a list; the output list is often composed of elements from the input list. The abovementioned function "z",for instance, if given the input "(a,b,c,d,e)", composes the output "(c)" from it. The function *rest*, if given any non-empty list, composes or 'returns' its output list by simply removing the first element from it input list. The function *id* returns even its input list quite unmodified.

Functions are also called 'transformations'. Input lists are also called 'argument list'. If an argument list contains only one element, the argument-list parentheses can be omitted.

Numbers and other constants can be regarded as functions that ignore their argument lists and return their own names instead. Thus, if we submit "(a,b,c)" to 5, we obtain the output "(5)". All names are constants unless we have given them an explicit function definition. When names of functions with an explicit definition are to be mentioned without being 'function calls', they have to be quoted. For instance, assume that we wish to print the *letter* "x"; we need to write "print('x)", because "print(x)" would involve the evaluation of the *function* x (with the result *nil* because x has no arguments here).

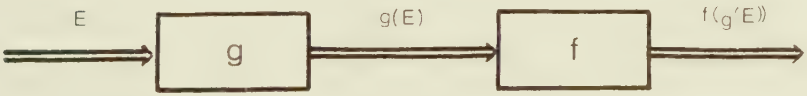
19.2. The REC Semantix

We shall now introduce the basic REC types of function meaning and simultaneously present the flowchart notation for REC. (Our presentation is based upon Tholerus 1972/5 and upon lectures held by Torgny Tholerus in 1973.)

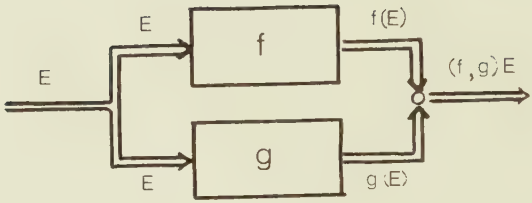
We said that a function f transforms an input list E into another list. We can symbolize this by

The arrow labels can be omitted in practice. We mean such a transformation, wherever we write "f" in a program.

Transformations can be coupled in various ways. Serially, for instance. Serial transformations can be written in the form "(f(g))E" and are understood to mean "(f(g(E)))", which can be symbolized by the chart

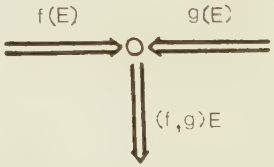


'Parallel' transformations can be written in the forms "(f,g)E" or equivalently "f(E),g(E)", and symbolized by

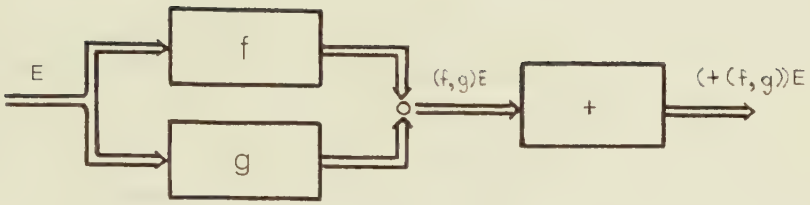


or simply

The result of a 'parallel' transformation (f,g)E is constructed as follows: take the first element of f(E) and attach it in front of g(E). This operation is symbolized by



For instance, "f+g" comprises three functions, namely f, g and + . f and g are to be evaluated 'in parallel' and their results subsequently submitted to +. In flow-chart form:



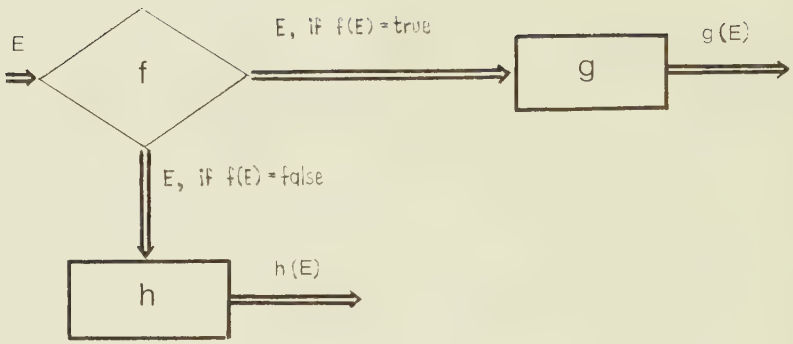
The detailed flow-chart symbolizes thus the *meaning* of "f+g" by showing the structure of the involved evaluations. This structure can in turn be reconstructed from "(+(f,g))E".

Three 'parallel' functions are, of course, called by writing "(f,g,h)E" or "f(E),g(E),h(E)".
(f,g,h)E = (f(g,h))E. Et cetera.

(Parallelity is, of course, only simulated. In reality, when we write "(f,g)E", f will be evaluated before g is evaluated. Side-effects of the evaluation of f can therefore influence the evaluation of g although g is applied to E. This is most drastically seen if we let f have the redefinition of g as a side-effect.)

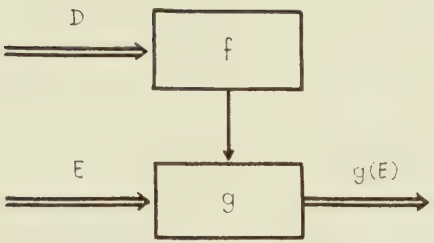
'Parallel' function calls can be used for augmenting a list. For instance, assume that we wish to attach the number 2 in front of the list "(a,b,c,nil)". We can do this by writing "(2,id)(a,b,c,nil)".

Conditional transformations can be written "(if f then g else h)E" or synonymously " if f(E) then g(E) else h(E)", and symbolized by

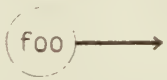


We do not always wish to apply another function to the output of a function. For instance, assume that we wish to compute a formula and then print the result, then read new arguments. The second step is obviously dependent upon the first, but the third is not dependent upon what has been printed. That means that "read(print (formula))E" would be incorrect, because we do not want to read what we have printed. We must instead write "print(formula))E; read". The semicolon indicates that the latest output list (here print's) is to be ignored from now on.

The semicolon indicates a program structure which we call 'progstructure'. We write "(f;g)E" or equivalently "f(E); g(E)" when we wish to specify that the result of f needs not be saved. We symbolize "f(D); g(E)" by

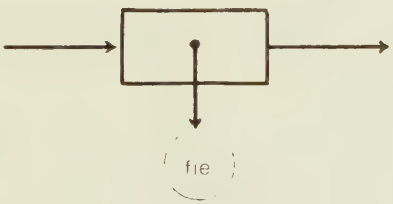


The simple arrow symbolizes succession in the program control flow when no output is to be 'transmitted'. Double arrows imply simple arrows, of course. Simple arrows can instead be used when we only wish to outline the control structure, irrespective of argument dependences.

Let 

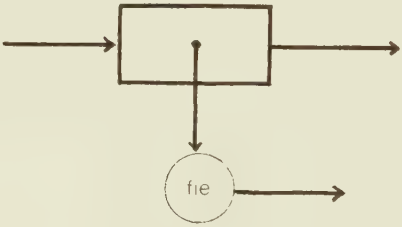
mean "the function foo is defined as follows:".

Let



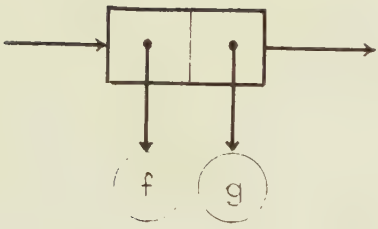
be equivalent to

We can thus represent meanings such as "apply the function *fie* which is defined as follows:" by

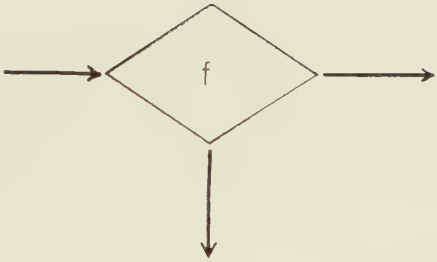


The function

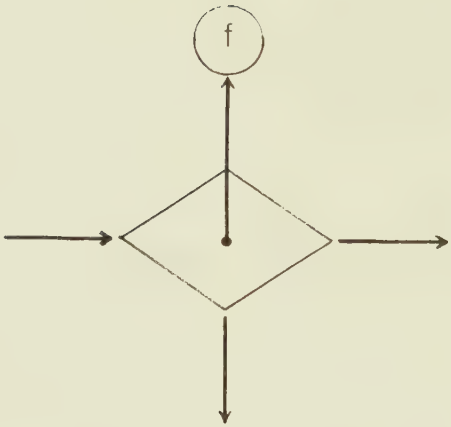
can, in analogy, also be written



and



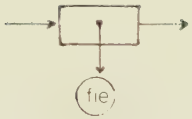
can similarly be written



The end of a function definition is indicated by "|".

19.2.1. Parentheses and Program Structure

Parentheses are used for indicating the desired program structure whenever it deviates from the basic form. Parentheses have



as their flowchart correspondence; this symbol means that control is to 'back up' to the departure point when fie has been evaluated. "fie" denotes here a parenthesized expression; it could also denote a function; a function is thus in principle a named parenthesized expression; the name is an abbrevation of the parenthesized expression.

This is important for our understanding of when to use parentheses around *if*-expressions (cf. section 20.2). Whenever we wish to continue at *one* certain point aften an *if*-expression, no matter how the flow went within it, we need to enclose the *if*-expression in parentheses.

20. Atman-REC

Atman3, in contrast to Atman2, contains only a single time queue for all agents. All jigs are made 'interaction jigs' (in the terminology of Atman2), that is: jigs are programs that can be suspended and which thus can describe processes that procede for a long time in parallel with other processes.

Torgny Tholerus has also 'stream-lined' the monitoring by advancing time not in minimal steps but directly up to the next scheduled moment of jig activation. Whenever a jig is to be suspended, we indicate for how long, and the jig will be scheduled for next activation immediately after the specified suspension period. *Indefinite* suspension is possible; we need only specify a condition to be fulfilled by some object (cf. section 20.3). This design is, of course, much more effective than the original one.

20.1. Jig Definition

Jig definition is easier than before.

Example (cf. section 12):

```
greet:instance attention (approx 20), priority 8;
     do print ("hello", x, " how do you do?")|
```

The words "instance" and "do" are operator names. *Instance* fetches and assumes the appropriate jig properties, while *do* initiates the evaluation of the jig body. The jig body consists of REC expressions.

20.2. Jig Sequences

Jig sequences can be made ordinary statement sequences since REC allows 'parallel' functions. However, we indicate jig 'parallelity' by using the operator "&", not the comma, since we are not interested only in the program-control structure created by it.

Parsing rules for '&': "f; g&H; i" is equivalent to "f; (g&h); i"; "f&(g&h)" is equivalent with "f&g&h" and also with "(f&g)&h"; "f; g&(h; i); j" is not equivalent with "f; g&h; i; j".

Example (cf. section 14):

```
...; rise; (if is-maid then milking else
            if is-female then prepare breakfast
            else inspect-stables); breakfast;...
```

is a part of a jig body, and implicitly a jig sequence.

All jigs referred to in a jig sequence are in principle assumed to belong to the sequence. In order to detach a jig from the jig sequence (cf. educationjig in Volume 4, section 10.2, where subtasks are to procede in parallel with the ancestor jig), we need therefore write "pushjig f". For example, if

```
...; e; pushjig f; g&h; i;...
```

is a part of a jig body, and e, f, g, h and i are jigs, f cannot be pushed before e has been abandoned, but g and h can be pushed as soon as f has been pushed; i, however, cannot be pushed before

the 'parallelly' computed jigs g and h have both been abandoned.

A 'competitive' parallel pushing of jigs can be achieved by using the pushing function 'xorpushjig'. (The prefix "xor" alludes to "exclusive or".) This function is used to push several jigs simultaneously, but the jig that is executed first is to suspend the other ones until it is abandoned (cf. Volume 4, section 10.2), whereafter the same rule applies again, et cetera.

20.3. Execution Suspension

Assume that we wish one link in a jig sequence to have a long duration. Atman2 would have checked many times whether the jig's duration had expired. Atman3 checks not at all; it places instead a signalling task into the time queue which is to activate the next jig in the sequence. Since every jig body is in itself a jig sequence, it is ingeniously simple to suspend jig execution. It is even possible to let the signalling task be conditional.

Suspension is commanded by writing

```
...; wait n;...
```

or equivalently

```
...; suspend jig, n;...
```

where n is an expression that evaluates to a number. "jig" is used in Atman3 to refer to the presently executed jig instance. The second expression is more general; we can namely replace "jig" by any other reference to a jig instance and thus suspend other jig instances.

If we wish to await a certain event m, which is to be caused or experienced by M, we write

```
...; wait for m;...
```

M is given a 'signal table', and an entry in this table that 'jig' is to be activated in case and when M causes or experiences m. m is usually specified in the form of a condition.

21. Syntax Extensions

21.1. Argument Indicators

Atman-REC allows us to define jigs as operators with 'argument indicators'. This is, in a way, a return to formal parameters, even a doubling of them. But since REC is variable-free, it is especially easy to implement this particular feature in it.

Assume a function call "move (a,b,c,d)", where a is an object, b the present location of a, c is the location to where a is to be moved, and d is a condition to be fulfilled. If we have many such functions, it can be difficult to remember arguments orderings. We wish also to omit a and b sometimes, because they are obvious, and d because there is no condition. In fact, we wish to be able to write "move object a from b to c if d" at one time, "move to c" at another. The orderings should also be variable; "move from a to b" should be synonymous with "move to b from a".

To allow such flexible notation, we need merely to define an operator '::' which automatically defines how the whole expression is to be transformed into its canonical form. The function will be associated with a small table which specifies the envisaged indicators (such as 'to' or 'from'); these indicators can, *within* the function body, be used for making references to the values they indicate.

That means that we merely need to write

```
move (from, to, object, if, by):: .... |
```

instead of

```
move: .... |
```

in order to obtain flexibility. The efficiency cost is negligible.

22. The Atman3 Database

It is important that the pointing to a particular item in the database is easy and of obvious and unambigious meaning for the program reader also, not only for the computer. We wish therefore to be able to write "the door of the house beside the house of John is green" instead of

```
"a:=tabx(John,'house); b:=tabx(a,'beside);
     c:=find(a,'house);
     if c then tabx (c,'door):='green".
```

We wish also to be able to ask for information similarly: "what is the color of the door of the house beside the house of John" and "is it true that the door of the house beside the house of John is green". The first question requires that we have said earlier in some connection that 'green' is a color; for example, when we said "the car of the person named John has the color named green".

Is it a particular *fact* or a general *rule* that the door of the house beside the house of John is green? If we do not decide the question explicitly, Atman3 will assume that it is merely a fact, not a rule. But we can write "rule:the door ..." or "fact: the door..." in order to be explicit on this matter.

Our experience with the programming of jigs for Atman2 tells us that a major portion of all such programming consists of searching certain information items, or of pointing to and updating them. Programming will surely be more fun as well as more efficient if these things can be more conveniently specified.

The updating of items is uniquely easy. For example, let us increase the family of John by one person; we write "(x:=x+1) the size of the family of John". Note that "(x:=x+1)" is to be understood as a function which has "the size of the family of John" as its argument; this input evaluates the input list to (x:=x+1), while 'x' is the function that picks the first element from its input list. It is obvious that arbitrarily complicated updating rules (conditional ones, for example) are equally easy to formulate if we deal with a direct or an indirect item reference.

The difference between rule and fact is more obvious when we have propositions such as "every door of every house beside every house of John is green". Assume that John owns two houses and buys a third. If the above proposition expressed a fact, nothing follows from the house purchase, but if it expressed a rule, we are to change the colors of all doors of all houses beside the new house of John to green if they are not green already.

Propositions that contain "a" are often more difficult to interpret. Take "fact:every door of a house beside the house of John is green", for instance. If no stored information item can be found that represents a house of John, such an item is to be created. If there are houses owned by John, none fulfilling the stated condition, we have two possibilities at hand: if all houses are explicitly said to have doors that are red, blue, et cetera, but not green, we are to create a new house which fulfils the condition. But if we cannot ascertain that the condition is not fulfilled, we must register a *category* that comprises all houses beside the house of John; this category is associated with

the condition. Each time any member of that category changes or is given a door color, we must check whether we now are able to draw more definite conclusions that render the category registering superfluous. (Note that we thus again avoid the frame problem.)

If we do not wish to have a new house created but instead want to have one of the houses beside John's house change its colors, we must write "change: every door of a house beside the house of John is green". A random selection is made. If we want to specify more exactly which house we mean, we need to point to it directly or to a subset of John's neighborhood.

Such pointing is done with the help of the function 'find ... such that': "find every house beside the house of John such that it is white, is old, has owner named Jim or has owner named Joe".

We wish also to be able to store facts or rules about relationships. For example: "fact:the house of John is more green than the car of Jeannine". This implies the storage of the facts that John has a house, that it is green, that Jeannie has a car, that it is green, and that a comparation relation exists between the car's color and the house's. More general facts will also be stored (if they are not already, of course), namely that persons can have houses and cars, and houses and cars can have colors

"rule: a house of a person is more large than a house of a dog" implies similarly that dogs can have houses. Adjectives are treated as names, belonging to a *sort*: 'green' was specified to be the

name of a 'color'; 'large' must simultaneously be specified to be the name of a 'size'. (This can be done by writing "(large has sort named size; large)" instead of "large" in the rule above.) Such information will be valuable when Atman3 is to try to 'solve problems'.

The above rule does not refer to concrete houses, persons, or dogs, but to categories. The comparation relation holds between the two categories 'a house of a person' and 'a house of a dog'.

23. Hypothetical Reasoning

Bobrow and Wegbreit (1973) have proposed an implementation design for the control of multiple database versions that allows a programmer to work effectively with several database versions simultaneously. Torgny Tholerus has already augmented REC with this new control structure, and we can now expect to be able to let simulated agents perform hypothetical reasoning.

Let us exemplify the power of the new control structure with a description of how the eight-queens problem (cf. Bobrow/Wegbreit 1973, p 595) can be solved: The task is to place "eight queens on a chessboard in such a way that no two can take each other". The function 'conflict' which we have omitted here, checks whether a queen in square s can take any of the already positioned ones.

```
n:=0; s:= nil;
loop: k:= select (1--8)
     check conflict (s,1);
     n:=:n+1; s:=:par(k,s);
     if n=8 then s else loop |
```

The operator ":=:" indicates (and sees to it) that the assignment is reversible in case backtracking becomes necessary.

(Note that the eight-queen problem can be regarded as a simple paradigm for role-taking in a social context.)

PART IV: DATABASE GENERATION

24. Database and Morphologies

It is important for clarity to distinguish between a dynamic model's set of states and its set of operating rules. The set of states we are interested in is represented (described) in a database. In our case, the database will always contain the description of only a single state. 'Morphologies' are structures within the database; intensional definitions of these structures can be used to generate them, and thus to generate the database.

The study of morphologies in the social world is useful for our possibilities to describe possible **total** social-world states precisely and concisely. We envisage the development of a powerful database monitor which accepts bits and pieces of an extensional database definition, and which, with the help of the intensional database definition laid down in the morphological rules can deduce ('generate') a complete database.

We have no such general monitor as yet, merely a few ad-hoc functions which generate a database in a less elegant manner. But it does no longer seem difficult to realize such a monitor. (Cf. ch.22.)

This morphology monitor does not in any way compete with the jig monitor (which specializes on the monitoring of the operating rules); in fact, the morphology monitor can be removed when the atman monitor starts its activity, unless we wish to rearrange the database in the middle of a simulation run according to specific morphological rules.

25. The Kinship Morphology

The following morphology is a good illustration of what we have said about a concise rule being able to generate a database, if we only have a monitor that can be instructed how to interpret the rule. We shall also use it as an example for how one must try to *find* a structure in the data on a given part of reality.

We wish to find a concise way to describe graphs of kinship relationships, such as

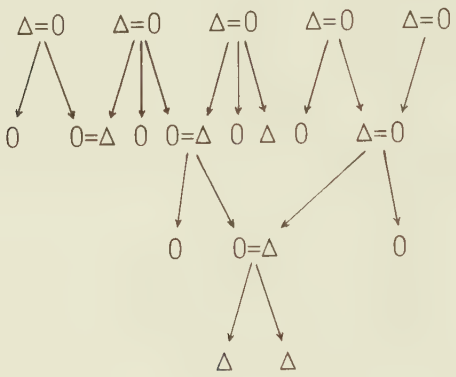


figure 2.1

Since such graphs may comprise hundreds or even thousands of elements, it is out of the question to specify them explicitly in every detail. We have found that the following rule enables us to generate such graphs:

```
Morphology 'Kinship:
     G4man ← 1 G3family;
     G4woman ← 1 G3family;
     G3man ← 1 G2family, 0--1 G3family;
     G3woman ← G2family, 0--1 G3family;
     G2man ← 1 G1family, 0--1 G2family;
     G2woman ← G1family, 0--1 G2family;
     G1man ← 1 G0family, 0--1 G1family;
     G1woman ← G0family, 0--1 G1family;

     G3family → 1 G3man, 1 G3woman, 0--2 G4man,
                0--2 G4woman;
     G2family → 1 G2man, 1 G2woman, 0--2 G3man,
                0--2 G3woman;
     G1family → 1 G1man, 1 G1woman, 0--2 G2man,
                0--2 G2woman;
     G0family → 0--2 G1man, 0--2 G1woman!
```

"G4man" means "a male person in generation 4", et cetera.
"0--2" means "choose a random integer value between 0 and 2, including 0 and 2".

The meaning of the arrows is less easily specified, and we have not yet found a good notation which we could use for communication with the envisaged morphology monitor.

Let "supra(x)" denote the set of families to which x belongs, and let "infra(y)" denote the members of the family y. "G3woman17" denotes a concrete female person of generation 3 in the database.

Let "x←ny" mean "if supra(x) does not contain an element which is an instance of the class y, and if n is not zero, create such an instance, z, and let x belong to infra(z)".

Let "x→ny" mean "infra(x) contains already m instances of y; if n exceeds m, create n-m additional instances of y and let them belong to infra(x)".

Finally, let "$x \leftarrow n_1 y_1, n_2 y_2$" simply be an abbreviation for the two rules "$x \leftarrow n_1 y_1$" and "$x \leftarrow n_2 y_2$".

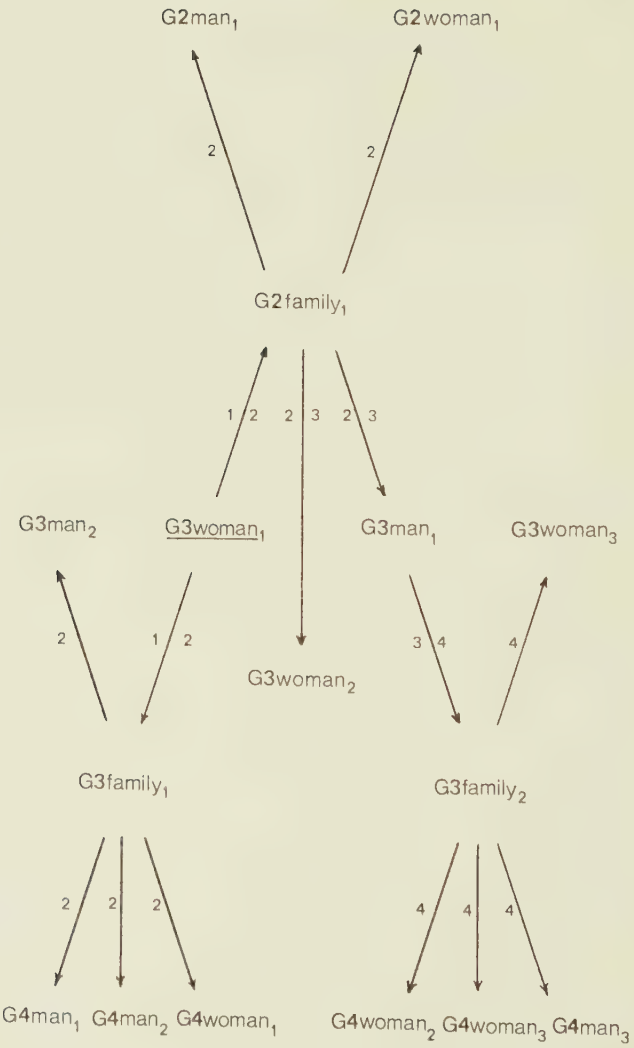
The generation may start with any person or family, G3woman1, for example. The fourth rule applies to this first instance of a G3 woman. Supra(G3woman1) is, as yet, empty, and we are thus to create one instance of a G2 family, namely G2family1, and if "0--1" happens to evaluate 1, we are also to create an instance of a G3 family, namely G3family1. Furthermore, we are to file G3woman1 into infra(G2family1) and into infra(G3family1), and G2family1 and G3family1 must be filed into supra(G3woman1).

We wish to avoid a 'depth-first' generation of this graph, because these rules tend to generate an infinite graph. We must therefore stop the generation by monitor intervention, and when we stop, we wish to have a net-like graph, not a string-like one. We impose therefore the monitoring rules that persons and families are focused in turns, and when we wish to stop generation, we do so in a 'natural' fashion by forcing all random choices in the morphology rules to result in the lowest possible value (that is "0--1" and "0--2" will then always evaluate to zero).

No other persons than G3woman1 exist as yet in the database, and we can therefore shift to focusing upon the families in the database. The tenth morphology rule applies to G2family1. Infra(G2family1) contains no instance of a G2man; G2man1 is therefore created and added to it. It contains no G2woman instance, and G2woman1 is therefore added.

Let the first occurrence of "0--2" evaluate to 1, the second to 2. There are not sufficiently many G3men nor G3women; G3man1 and G3woman2 are therefore added. And we 'expand' G3family1 similarly. When both families have been considered, we return to the now already much larger set of persons and 'expand' them. Et cetera. Figure 2.2. illustrates how the graph grows. The numbers by the arrows indicate during which 'pass' of rules it was created or tested. Passes 3 and 4 are not recorded completely in order to keep the figure surveillable.

figure 2.2


G2man1
G2woman1
2
2
G2family1
1 2
2 3
2 3
G3man2
G3woman1
G3man1
G3woman3
2
1 2
3 4
4
G3woman2
G3family1
G3family2
2
2
2
4
4
4
G4man1 G4man2 G4woman1
G4woman2 G4woman3 G4man3

26. Attributes

While we create this database, we can also assign certain attributes to the created entities, in order to be able to identify them more easily later on. The attributes we think of are 'is husband in F', 'is wife in F', 'is mother of (x,y,z)', 'is child in F', 'is person', 'is family', 'is male', 'belongs to generation 2', for example.

26.1. Ages

When all entities have been created, we can try to assign ages. This is not at all easy, it seems, but we employ as yet the following rules:

Start with all families belonging to generation 3 and assign 'age modes' 0--17 to them. Let "m" denote the obtained age mode for a particular family F3. All children in F3 are assigned an age m3±(0--4) (but not less than 0). (The relatively high proportion of twins can be reduced by letting child-age assignment be Markovian.) The husband in F3 is assigned the age h3:=m3+(26--36). The wife in F3 is given the age w3:=h3+2-(0--7). This means that in the youngest families will the ages have been 17 for the mother and 22 for the father when the first child was born. In the oldest families in the third generation the mother will be 55, the father 53, and the oldest child 21.

Consider next the father of the wife in F3 (unless he already has an age) and assign the age h2:=w3+21--30 to him. His wife is given the age w2:=h2+2-(0--7), but w2-w3 may not be less than 17. The age mode of F2, the family of origin of the wife in F3, is set to m2:=w3. All children in F2 who not

yet have been given an age, obtain one according to c2:=m2±(0--4), but observing the rules that h2-c2 may not be less than 20, and w2-c2 not less than 17.

The ages in the family of origin of the husband in F3 are assigned according to the same rules. All grandchildren of F2 are then given ages in the appropriate ranges (if they do not have ages already, of course). The families of origin of the husband and wife in F2 are then treated similarly, et cetera.

26.2. Social Status

We assign social-status values to families next. We start with the families in generation 3 and give them random values according to the scheme

(1/2:3; 3/4:2; 1/1:1),

which means that half of all families obtain social status 3, three fourths of the other half obtain status 2, and the rest obtains status 1.

Let "a→(b,c),(d,e)" mean that if a family F has social status a, then will the family of the husband either have status b or d. If it has b, then will the family of the wife have status c, otherwise e. The picking of either pair (b,c) or (d,e) is at random and with equal chances.

We assume the following scheme:

3 → (3,3)
2 → (3,3),(2,3),(3,2),(2,2)
1 → (2,2),(2,1),(1,2),(1,1)

Status for families of generation 1 and 0 are assigned similarly.

Several status assignments can be attempted if a family has several married children, but only the first one is performed.

26.3. Attribution in Atman2

The 'morphology monitor' contained in the CREATE package of Atman2 is not flexible enough to enable us to submit attribution rules such as the ones specified hitherto. Although we therefore cannot keep it, let us take a short look at some of its features.

We used the function EQUIP for generating property trees or parts of property trees. Each hierarchy level in the tree was indicated by the sign "/". For instance,

```
(EQUIP EACH PERSON WHICH-IS SON
     WITH INDICATORS
          (/ SEX MALE AGE 20 ))
```

is a rule which assigns the sex 'male' and the age 20 to each person who is a son. This example illustrates that we need a possibility to assign random values, because it is unrealistic to assume that all sons are aged 20. The rule

```
(EQUIP EACH PERSON WITH INDICATORS
          (/ AGE (RAND 1 99 )))
```

assigns random values between 1 and 99 to the property *age*. Lists can be generated by writing

```
(RAND (L1 U1) (L2 U2) ....(Ln Un))
```

where each pair (L U) evaluates into a random numbet between L and U.

Other instructions that are evaluated to render concrete values are:

(EVAL <form>)

where <form> is a LISP expression; in particular:

(EVAL SELF)

evaluates to the name of the currently focused entity; and

(ANY x1 x2 ... xk)

which selects at random exactly one of the alternative values x1,...xk.

(EVAL WAYUP)

evaluates to the reverse list of indicators via which we can arrive at the property-tree position where this instruction's result is to be stored. For instance,

```
(EQUIP EACH PERSON WITH INDICATORS
     (/ SEX (ANY MALE FEMALE)
        NAME (EVAL SELF)
        LOCATION (/ GEOGRAPHICAL (RAND (1 100)(1 100) )
                    AGE (RAND 1 99)
                    TEST1 (/ TEST2 (EVAL WAYUP) ))))
```

could result in a property tree for Person1 such as

```
(/ sex 'male
  name 'person1
  location (/ geographical '(35 71)
              age 19
              test1(/ test2 '(test2 test1 location
                              person 1)))).
```

Such lists of indicators are useful when one at one position wishes to refer to properties at other positions in the property tree.

(EXPLODE <form> s1 s2 .... sk)

can be used for generating several subtrees. <form> is an expression which evaluates to a list of values which are used as indicators on the next (lower) tree level, while s1,..., sk are fixed values or instructions of the types we have described. For instance,

```
(EQUIP EACH PERSON WITH INDICATORS
     (/ ACQUAINTANCES
          (EXPLODE (SAMPLE 3 (EXCEPT (EVAL SELF)
                                     (GETALL 'PERSON)))
                   (RAND 1 100) (ANY POS NEG) )))
```

could for some person result in the subtree

```
     (/ acquaintances
          (/ person2 (35 'neg)
            person8 (78 'pos)
            person12 (19 'pos)).
```

Several EQUIP instructions can be executed after each other. Several property trees can thus be superimposed upon each other.

27. Physical and Social Morphology

We shall not specify the physical or social morphologies in detail and exact notation, because we are still working on them and also currently trying to invent a new database design (see chapter 22). The rule notation specified in Volume 1, chapter 3, can at present be used for database specification when we wish to use Atman2; we hope that no special notation will be necessary in connection with Atman3, but that atman-REC will be powerful enough to imply such a notation.

We shall instead focus upon the kind of information that we expect to submit to Atman as an intensional definition of the database we wish to work on. The exact operational details of the rule use are important for how much information must be explicitly supplied and how much can be derived, but we are unfortunately as yet unable to report these details. However, we expect that a not much greater amount of explicit information needs be given than is contained in our specifications below, except what concerns informal social structures.

We use here an ad-hoc notation which relates to the notation explained in Volume 1, chapter 3, as follows:

We write

```
farm→1 family; 1--2 person(farmhand);
      1--2 person (maid)!
```

instead of

```
(*Each farm resolves into 1 family,
      1-2 person with-status farmhand,
      1-2 person with-status maid)
```

We write

```
institution→(1/10 : 1 library;
             3/9  : 1--3 classroom;
             5/6  : 1 cultural-institute;
             1/1  : 1 political-institute) !
```

in order to 'resolve' given insitutions into libraries (in one tenth of all cases), into 1--3 classrooms (in three ninths of the remaining cases), et cetera. If the last alternative does not have the probability 1, and is not chosen, we restart from the top, unless there is only a single alternative.

```
a→(1/2: 1 b); (1/2: 1 c)!
```

gives us thus a different result than

```
a→(1/2: 1 b; 1/2: 1 c).
```

We ignore here the details of how this notations is to be translated by the database generator. Each morphology-rule set requires in principle a specific interpretation, which we outline only verbally here:

The morphology-rule set PHYSICAL generates physical contexts:

Morphology-Rule Set PHYSICAL

```
region → 2--5 urbanized-area,
         5--10 rural-area;

urbanized-area → 2--5 middle-sized (autonomous),
                 0--1 metropolitan,
                 4--6 small-town;

rural-area → 2--5 cultured-area,
             1--2 special-area,(1/10: mining-area),
             1--5 natural-area;
```

```
metropolitan → 1 city, 0--1 special-area,
               3--10 suburb, 1--5 industrial-area;

middle-sized →6--10 residential-area (varied),
              1--3 industrial-subarea,
              (autonomous: 1 town-center),
              (satellite: 1 suburb-center);

suburb → 1 middle-sized (satellite);

small-town → 1--3 residential-area (low-rise),
             0--1 industrial-subarea,
             0--1 rural-environment,
             1 town-center;

city → 1 city-center, 1--3 administrative-area,
       5--10 residential-area (high-rise),
       5--10 city-industrial-area;

city-center → 1--3 town-center;

town-center → 3--15 house (special),
              5--10 house (office),
              1--5 deptstore-building,
              1--3 park, 1 railway-station,
              1--3 square;

suburb-center → 3--10 house (special),
                1--5 house(office),
                1--3 department-store, 1--3 park,
                1--3 lot, 1 square,
                0--1 railway-station;

administrative-area → 1--15 block (office);

block(office) → 5--10 house (office);

house(office) → 5--15 office-suite, 0-15 shop;

office-suite → 1--10 office,
               (1/10: 1 office-landscape);
```

```
residential-area (varied) →
                0--3 residential-area(low-rise),
                0--3 residential-area(medium-rise),
                0--3 residential-area(high-rise);

residential-area(low-rise) → 5--10 block(low),
                0--5 block (medium), 1--3 lot,
                1--3 park;

residential-area(medium-rise) → 0--1 service-center,
                0--5 block(high), 5--10 block(medium),
                0--5 block (low),
                0--1 park, 0--1 lot;

residential-area (high-rise) → 0--2 service-center,
                5--10 block (high), 0--5 block(medium)
                0--1 park, 0--1 lot;

block(high) → 5--10 house (many-stories),
                0--1 park, 0--1 lot;

block(medium) → 5--10 house(several-stories),
                0--2 park, 0--2 lot;

block(low) → 5--20house(villa), 1--5 park, 1--5 lot;

house(several-stories) → (1/10: 1 shop),
                (1/10: 1 workshop), 2--15 apartment;

house(many-stories) → 20--50 apartment,
                (1/10:1--10 shop),(1/10: 1 workshop),
                (1/10:1--5 office-suite);

house(villa) → 1--2 apartment;

service-center → 0--3 house(special), 0--1 house
                0--2 house(office),0--1 departm.store,
                0--1 park, 0--1 lot;

railway-station → 1 railway-terminal (goods),
                1 railway-terminal(persons);
```

```
special-area → (1/2: military-installments),
               (1/6: supermarket-area),
               (1/6: airfield),
               (1/6: prison-area);

industrial-area → 1--10 industrial-subarea,
               1--2 goods-terminal,
               1--5 lorry-terminal-area,
               1--10 warehouse;

industrial-subarea → 5--10 mill-complex,
               1 harbor-area;

city-industrial-area → 2--10 mill;

mill-complex → 1--10 millhouse, 0--1 house(office),
               1--3 park, 1--3 lot;

millhouse → 1--10 work-station (industry);

mill → 1--10 work-station (industry);

goods-terminal →1--10 work-station (transport);

lorry-terminal-area → 1--10 work-station(transport);

harbor → 1--10 work-station (transport);

workshop → 1--10 work-station(industry);

cultured-area → 1--5 village, 0--5 smallfarm,
               5--10 farmstead, 0--1 estate
               (1/10: 1 manor);

natural-area → 0--3 forest, 0--3 lake, (1/10: 1 lot),
               0--3 grassland;

rural-environment → 10--50 farmstead;
               0--20 smallfarm;

village → 1 village-center, 1 rural-environment;

village-center→5--20 rural-house, 0--3 house(special);
```

```
farmstead → 2--3 rural-house, 2--5 barn/stable,
            2--8 arable land, 2--5 meadow,
            2--3 forest,(1/10: 1--3 lake);

smallfarm → (1/2: 1 croft), (1/2: 1 small-holding);

croft → 1 rural-house, 1 barn/stable,
            1--2 arable land, 1--2 meadow,
            0--1 forest;

small-holding → 1 rural-house, 1--2 barn/stable,
            2--4 arable land, 2--4 meadow,
            0--2 forest;

estate → 5--15 rural-house, 5--25 barn/stable,
            5--40 arable land, 5--25 meadow,
            5--15 forest, (1/10: 5--15 lake);
            1--5 lake, 0--15 smallfarm;

manor → 2--10 estate;

rural-house → 1--3 apartment, 1--3 park |
```

The generated physical contexts are differentiated with the help of the following rules which specify 'overlaps' with social contexts (which here are predominantly economic-purpose organizations):

Morphology-Rule Set SOCIAL

```
deptstore-building → (1/1: department-store)

shop → (5/10: 1 local-shop,
        1/10: 1 local-bank,
        1/10: 1 restaurant/pub,
        1/10: 1 local-police,
        1/10: 1 club,
        1/10: 1 dance-hall,
        1/10: 1 bingo-hall);
```

```
supermarket-area → (1/1: 1 supermarket);

house(special) → (1/15: 1 central-bank,
                  1/15: 1 church,
                  1/15: 1 hotel,
                  1/15: 1 school,
                  1/15: 1 cinema/theater,
                  1/15: 1 public-office,
                  1/15: 1 museum,
                  1/15: 1 hospital,
                  1/15: 1 central-library,
                  1/15: 1 sports-hall,
                  1/15: 1 concert-hall,
                  1/15: 1 parking-house,
                  1/15: 1 town-hall,
                  1/15: 1 police-headquaters,
                  1/15: 1 law court);

office-suite → (4/8: civil-office,
                2/8: bureau,
                2/8: doctor/nursery/dentist,
                2/8: local-library);

prison-area → 1 prison;

military-installments → 1 barracks;

lorry-terminal-area → 1 lorry-terminal;

goods-terminal → 1 railway-terminal(goods);

harbor-area → 1 harbor;

airfield → 1 air-terminal;

warehouse → (1/4: construction-office,
             3/4: storage);

lot  → (1/20: construction-site,
        1/1:park);

mill  → (1/50: power-plant,
         1/1: manufacturing);
```

millhouse → 1 manufacturing;

estate → 1 manoring

farmstead → 1 farming;

apartment → 1 household |

Morphology-Rule Set ORGANIZ:

We expect to evaluate thus rule set in several passes. We must first decide how many, and then which, basic contexts are to be dependent or autonomous in the various sectors. We make then appropriate aggregations, minimizing geographical dispersion, for instance. We ask then the same questions once more with regard to the entities that have resulted from the fusions. We do not yet know how to attribute formal ownership of such entities to persons, however.

Observe that all aggregations also imply the generation of supervisor and/or coordinator positions. The rule set looks approximately as follows below. We specify only the structures for autonomous firms, but assume the following correspondences:

Autonomous entities	Corresponding dependent entities
mini-firm	work-station
small-firm	department
medium-firm	division
large-firm	——

We shall seek to observe that there are hybrid firms such as small-firm chains where the elements (the small firms) are fairly autonomous but obtain advice from a common coordinating unit. There may similarly be minifirm chains and in some cases also medium-firm and large-firm chains ('trusts' and multinational firm-complexes). Some firm-complexes may consist of a mother firm and a multitude of small daughter firms, sales offices in different countries, for instance.

This is our preliminary rule set:

administration ⇒ 1 coordination,
1 purchase,
1 sales,
1--2 economy,
1 personnel,
1 research/development;

inventory ⇒ 1 raw-material,
1 semi-finished,
1 finished;

these rules define differentiations to be automatically made in the subsequent rules when possible; if the number of distinguishable entities is less than required for a proper differentiation, no differentiation is made:

mini-firm → 1 unit (administration),
1 unit (production),
1 unit (inventory);

small-firm → 1 group (administration),
1 group (production),
1 group (inventory);

```
medium-firm → 1 dept (administration),
              1 dept (production),
              1 dept (inventory);

large-firm → 1 division (administration),
             1 division (production),
             1 division (inventory);

unit → (production: 1--10 work-position;
        inventory: 0--2 work-position;
        administration: 1--3 work-position);

group → (production: 3--10 work-station;
         inventory, administration: 1 work-station);

dept → (production: 5--25 work-station;
        administration: 6--7 group;
        inventory: 3 group);

division → (production: 5--20 dept;
            administration: 6--7 dept;
            inventory: 3 dept);

work-station → 2--10 work-position |
```

The Morphology-Rule Set WPOS:

WPOS specifies an ordering amongst various types of work positions, and it generates these. Observe that these rules do not speak of *persons* but of *positions* (that is: of social and/or imaginary contexts):

```
estate →  1 estate-administrator;

estate-administrator → 1 foreman;

manor → 1 manor-administrator;

manor-administrator → 1 supervisor;
```

```
supervisor → 3--5 foreman;

foreman → 5--12 farmhand, 3--7 maid;

farmstead → (3/4: 1 peasant; 1/1: 1 petty-tenant);

peasant → (1/2: 1--3 farmhand, 1 maid);

smallfarm → (1/2: 1 crofter; 1/1: 1 smallholder);

hospital → 1 dept (level 6, administration),
           5--20 chief physician (level 6);

chief-physician → 3--10 physician (level 5),
                  1 chief-nurse (level 3);

chief-nurse → 6--40 nurse (level 2);

nurse → 3--10 assistant-nurse (level 1);

doctor/nursery/dentist→
          (1/2: 1--3 nurse (independent, level 3);
           3/4: doctor/dentist (level 5);
           1/1: 2--3 doctor/dentist (level 5));

doctor/dentist → 1--3 assistent-nurse (level 2);

dept → 1 dept-head;

dept-head → 5--6 group-head;

group-head → 1--5 clerk/worker,
             1--2 clerk/worker (level 1);

division-head → 1 personal-staff (level 3),
                3 dept-head;

company-head → 1 personal-staff (level 4),
               3 division-head;

personal-staff(x) → 1 private-secretary (level 2),
                    1--2 personal-assistant (level x);

small-firm → 1 president (small-firm);
```

medium-firm → 1 president (medium-firm);

large-firm → 1 president (large-firm)|

This set is not exclusive. For politics, education, and the military and police realms, for instance, other organization rules need be specified for assigning levels to workpositions. When only labels differ, we can use another morphology, called LABELEQUIV, for making labels correspond. For instance:

Morphology-Rule Set LABELEQUIV:

supervisor ↔ workposition (level 3);

estate-administrator ↔ workposition (level 3);

manor-administrator ↔ workposition (level 4);

president(medium-firm) ↔ division-head;

president(large-firm) ↔ company-head;

president(small-firm) ↔ dept-head;

foreman ↔ grouphead;

clerk/worker ↔ workposition (level 2);

group-head ↔ workposition (level 3);

dept-head ↔ workposition (level 4);

division-head ↔ workposition (level 5);

company-head ↔ workposition (level 6);

farmhand ↔ workposition (level 0b);

cottager ↔ workposition (level 0b);

maid ↔ workposition (level 0a);

smallholder ↔ workposition (level 1);

petty-tenant ↔ workposition (level 2);

crofter ↔ cottager

peasant ↔ workposition (level 3)|

The following rule set stipulates simply a fuzzy correspondence between work-position levels and incomes from occupations that belong to these levels:

Morphology-Rule Set WAGES/SALARIES:

Level 0a → 1500;
level 0b → 1750;
level 1 → 2000;
level 2 → 3000;
level 2a→ 3500;
level 3 → 4500;
level 4 → 6750;
level 5 → 10000;
level 6 → 15000;
level 7 → 22500 |

The next rule set assigns an attribute to each social context which tells which economic sector the context belongs to:

Morphology-Rule Set SECTOR:

primary ← agriculture, power,industry, mining;

secondary ← manufacture, construction;

tertiary ← trade, transport, service, welfare, education, health, finance, public-administration, control-force, mass-media, cultural;

power-industry ← power plant;

mining ← mining-site;

manufacture ← manufacturing;

construction ← construction-office,
construction-site;

trade ← deptstore, local-shop, supermarket;

transport ← lorry-terminal, railway-terminal;

service ← hotel, restaurant/pub, parking-house;

welfare ← welfare institution;

education ← school, university, research institute;

health ← hospital, doctor/nursery/dentist;

finance ← central-bank, local-bank,
insurance-agency;

public-administration ← town-hall, bureau,
public-office;

control-force ← barracks, prison, police-headqu,
law-courts, local-police;

mass-media ← radio/tv-station, newspaper, printing-
industry, publishing-house;

cultural ← church, museum, central-library, local-
library, sports-hall, club, dance-hall,
bingo-hall, concert-hall, cinema/theater;

agriculture ← manoring, farming |

We need also a set of induction rules which aggregate work positions into work stations, work-stations into units, units into departments and small firms, et cetera, but we are as yet uncertain how these aggregations are to be performed. They cannot be made in a random fashion; we need, for instance,

seek to minimize geographical dispersion.

A similar, but even vaster problem, is the final matching of persons created by the kinship morphology with work positions. But one sees that database generation according to intensional definitions (and possibly also a few extensional definitions) can be elaborated into a sort of 'static simulation' or pre-simulation that has nothing or little to do with the dynamic simulation that starts when the database has been generated. Database generation by 'static simulation' can be made a research region of its own within the atman-system research because it has social-science interest by itself.

28. References

Bobrow/Wegbreit 1973
Daniel G. Bobrow/ Ben Wegbreit: A Model and Stack Implementation of Multiple Environments. *In*: *Communications of the ACM*, *16*, No 10, Oct., 1973. (Re: 23.0.)

DLU74
Anders Beckman et. al.: *Datalogilaboratoruet 1974*. Uppsala: Computer Science Laboratory, Feb., 1974. (Re: 19.0; 19.1.)

IJCAI73
Advance Papers of the Third International Joint Conference on Artificial Intelligence. Stanford, Calif.: SRI Publ. Dept., 1973

Iverson 1962
K.E. Iverson: *A Programming Language*. New York: Wiley, 1962. (Re: 19.0.)

Risch 1972
Tore Risch: GETT/PUTT Programdokumentation. Uppsala: Computer Science Laboratory, 1972. (Manuscript) (Re: 10.0.)

Risch 1973
Tore Risch: MAKROEXPAND - ett Enkelt Makroexpanderingspaket i LISP. Research Report DLU 73/22. Uppsala: Computer Science Laboratory, Nov., 1973. (Mimeo.) (Re: 10.0.)

Tholerus 1972/5
Torgny Tholerus: Preliminär Dokumentation av REC. Uppsala: Computer Science Laboratory, May, 1972. (Mimeo.) (Re: 19.2.)

Tholerus 1972/10
Thorgny Tholerus: Funktionsteori. Uppsala: Dept of Philosophy, Oct., 1972.(Re: 19.0.)

Sociocybernetics

is a way to model social reality, and man in particular. It is an integrating interdisciplinary approach which explicitly acknowledges the fact that scientific capacity has been essentially augmented through the advent of the computer as an easily accessible tool. Sociocybernetics can be said to be a modern and operational epistemology for the social and behavioral sciences.

Homo Cyberneticus

is an exemplification of how sociocybernetics is to be used for comprehensive causal modelling. *Homo Cyberneticus* seeks to outline a coherent map of everyday life, upon which computer simulations of social-interaction processes can be based. It can also be looked upon as a research program that can systematize and unify research efforts in otherwise quite dissimilar fields, for instance psychology of religion and robotology, linguistics and ergonomics, or literature and futurology. It seeks to thus reassemble man into one whole.

Homo Cyberneticus comprises 5 booklets:

Volume 1: The Simulation of Everyday Life: A Synopsis
Volume 2: Towards an Epistemology of Dynamic Causal Modelling
Volume 3: Heuristic Simulation Programming
Volume 4: Artificial Psychology and Generative Micro-Sociology
Volume 5: A Macro-Sociological Framework

Please send your orders to your booksellers or to
SOCIOGRAFICA
Vikingagatan 50, S–753 24 UPPSALA, SWEDEN